RISC-V for Embedded Systems: A First Introduction

Dr. Adam Teman
Udi Kra

3 November 2019

Emerging Nanoscaled
Integrated Circuits and Systems Labs

Tradition of Excellence
Bar-Ilan University
אוניברסיטת בר-אילן

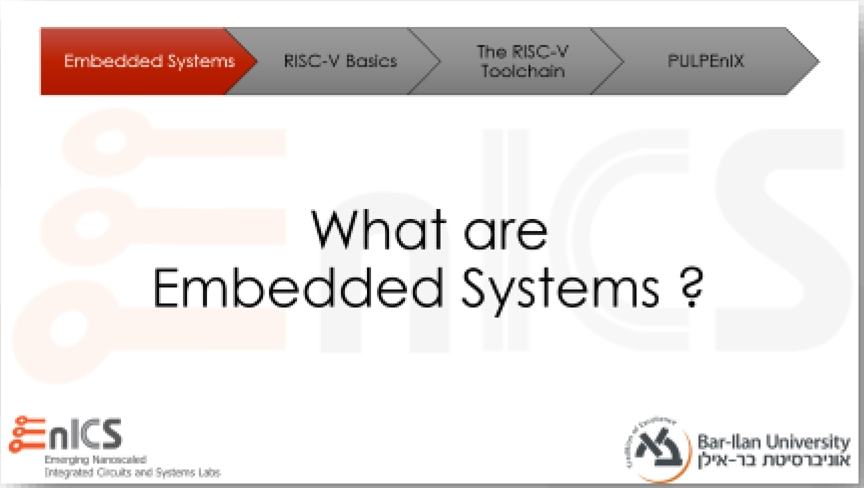
Who and why?

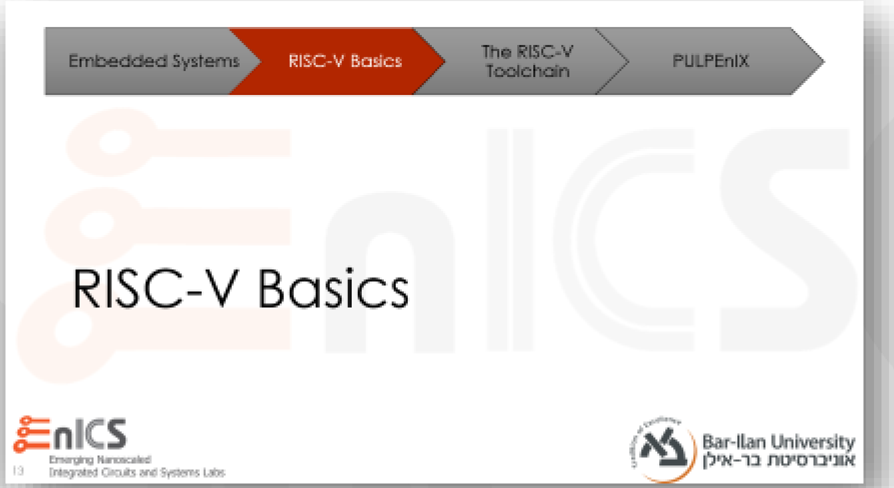
Emerging Nanoscaled
Integrated Circuits and Systems Labs

- The EnICS Labs Impact Center at Bar-Ilan University.
- Focal point of the GenPro Consortium, developing the Israeli RISC-V Platform.
- Developing the PULPEnIX SoC Platform, available for use in the Hackathon.

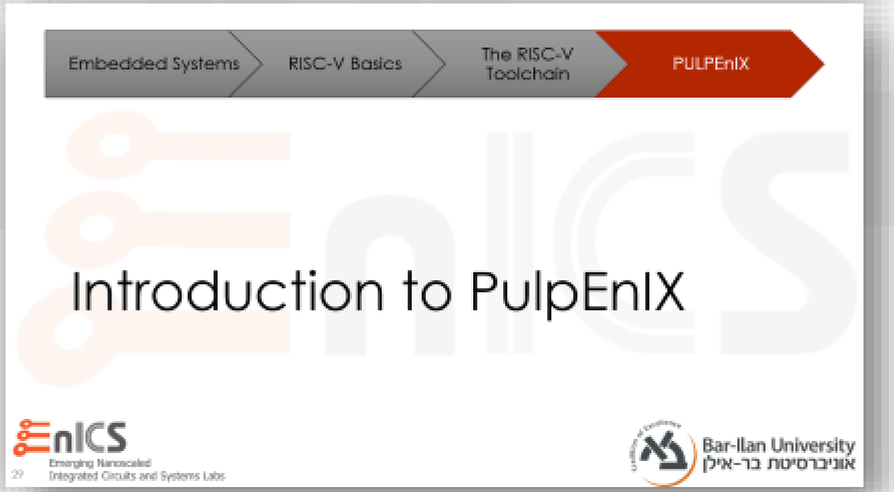
GenPro PULPEniX FPGA

Outline


Embedded Systems
RISC-V Basics
The RISC-V Toolchain
PULPEnIX
What are Embedded Systems ?
EnICS
Bar-Ilan University


Embedded Systems
RISC-V Basics
The RISC-V Toolchain
PULPEnIX
RISC-V Basics
EnICS
Bar-Ilan University


Embedded Systems
RISC-V Basics
The RISC-V Toolchain
PULPEnIX
The RISC-V Software Toolchain
CALL: Compiling, Assembling, Linking and Loading
EnICS
Bar-Ilan University


Embedded Systems
RISC-V Basics
The RISC-V Toolchain
PULPEnIX
Introduction to PulpEnIX
EnICS
Bar-Ilan University

What are Embedded Systems ?

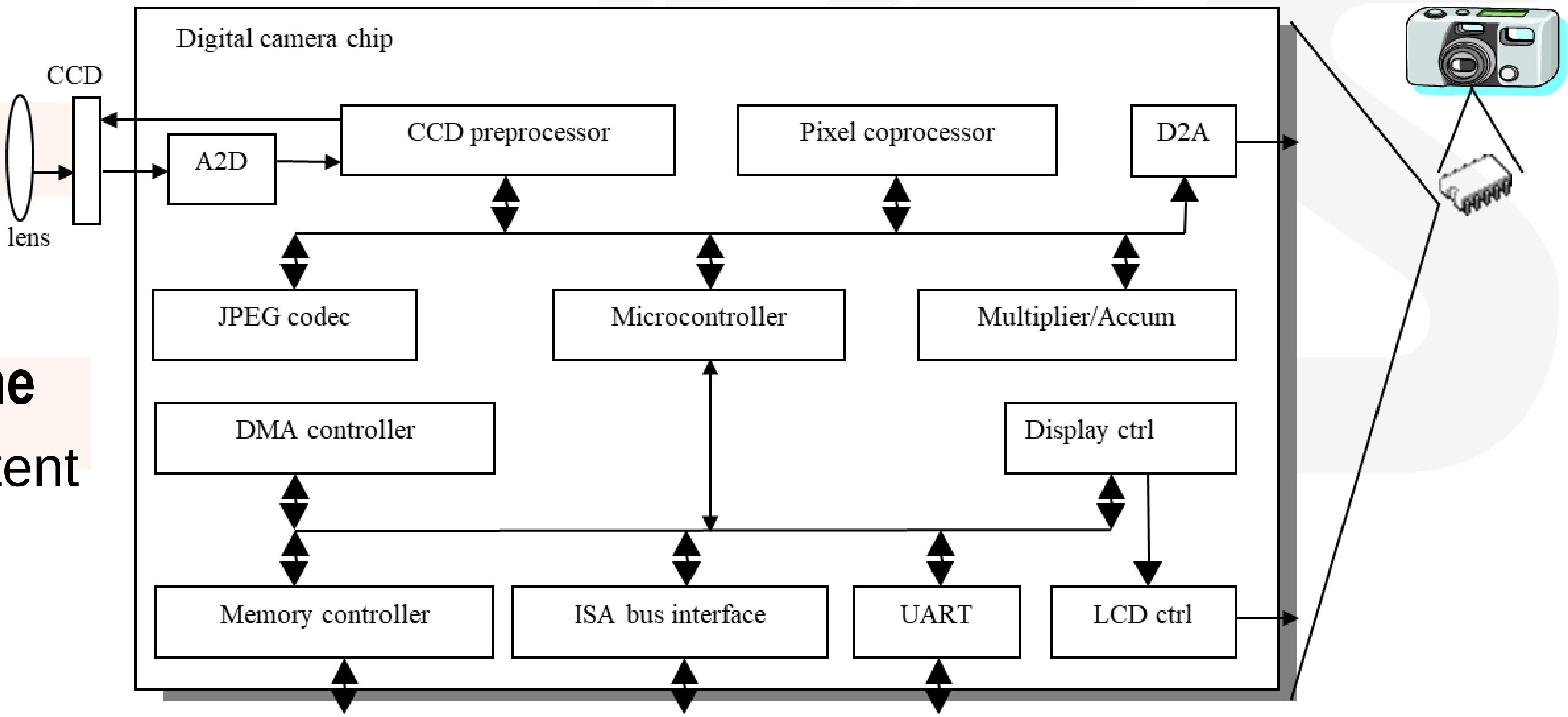
Embedded Systems

- **When we discuss computers, we usually think of desktops, laptops, tablets…**
- **But there's another *far more common* type of computing system:**
  - **The Embedded System**.
  - A computing system embedded within an electronic device.
- **An embedded system is a special-purpose computer system designed to perform one or more dedicated functions often in real-time**
- **Embedded Controller:**
  - No operating system ("Bare Metal"), small operating system (e.g., FreeRTOS), or perhaps full blown Linux-compatible.
  - Can code in high-level language (e.g., C) and compile, write directly in Assembler, or add task-specific hardware (accelerators).

Source: Embedded Systems Design: A Unified Hardware/Software Introduction

Embedded System Example – Digital Camera

- **Single-functioned**
  - Always a digital camera
- **Tightly-constrained**
  - Low cost
  - Low power
  - Small
  - Fast
- **Reactive and real-time**
  - Only to a small extent

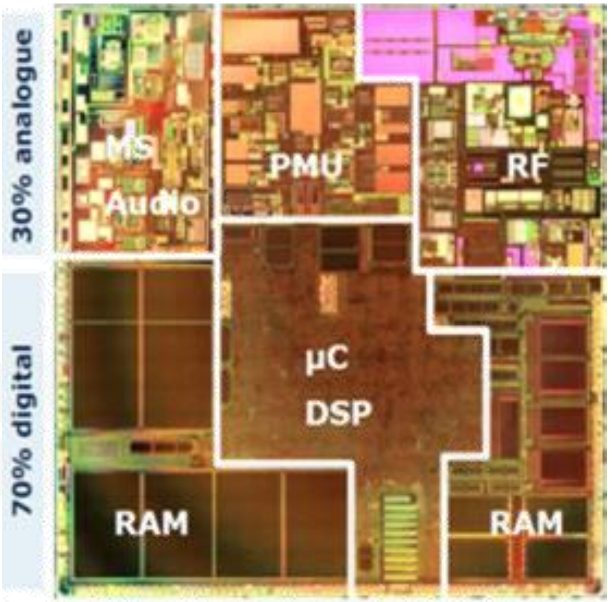


System-on-Chip (SoC)

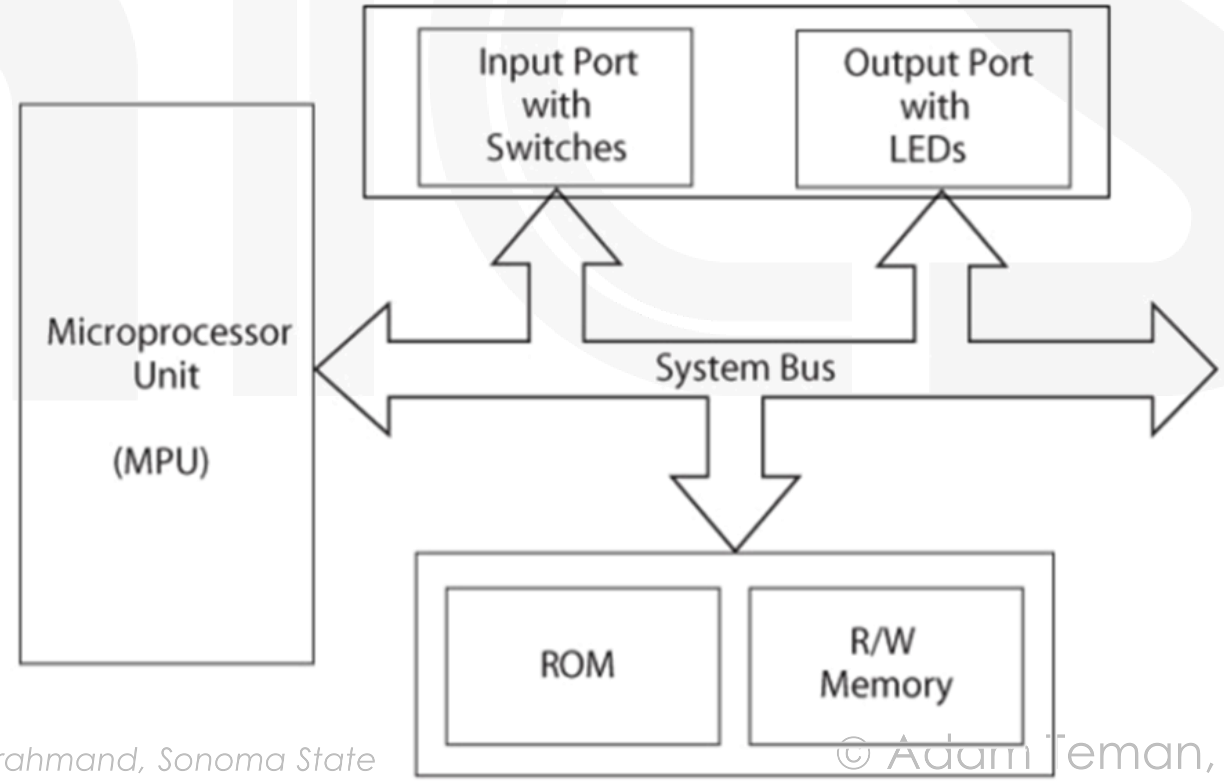
- **Embedded Systems are usually based on a central chip that includes:**
  - Microprocessor
  - Memory
  - Input/Output (I/O) circuitry
  - Buses
    - Address bus
    - Data bus
    - Control bus
- **Essentially, this is an entire system integrated on a single chip:**

A System-on-Chip (SoC)

Example:
Infineon E-GOLDVoice
"Phone-on-a-Chip"



Source: Raghunathan, ECE 695R

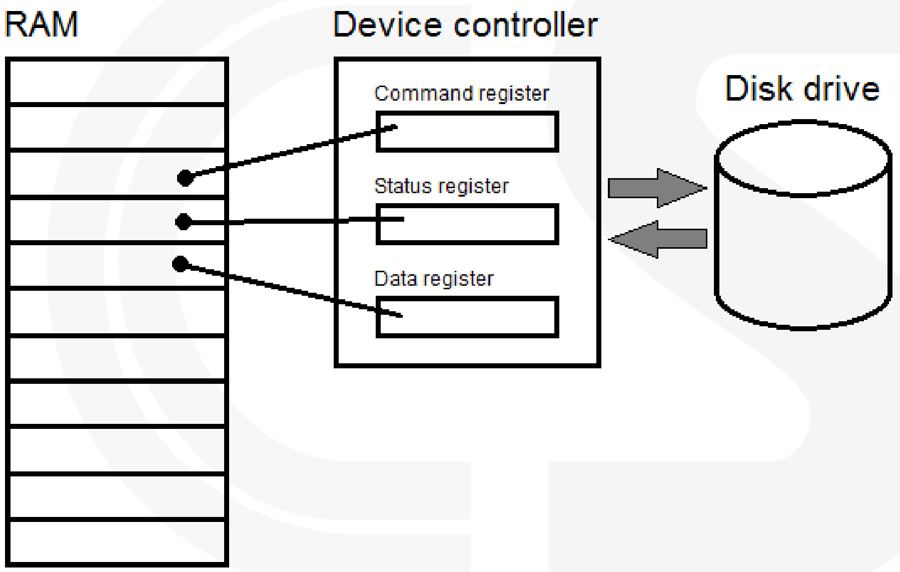


Source: Farahmand, Sonoma State

Memory Mapped I/O

Just an important concept that is sometimes missed by undergrad students…

- **Basically everything in an SoC is a memory address:**
  - Data is, of course, stored in memory.
  - Instructions are stored in the same memory space as data in a “Von Neumann Architecture”
  - And any peripheral or bus connection is just a memory address.
- **This is known as “Memory Mapping” or “Memory Mapped I/O”**



https://softwareengineering.stackexchange.com

  - There are no special instructions for controlling or accessing a peripheral.
  - Just write to an address or read from that address.
  - The spec provides a memory map that tells the programmer what is mapped to each memory address.

The Microprocessor Monopoly

- **The heart of the SoC is the microprocessor**
  - There may be several (even hundreds of) microprocessors on a SoC!
- **Only two main players in the microprocessor field or rather, only two Instruction Set Architectures (ISAs):**
  - **Intel "x86"**:
    - A complex instruction set (CISC).
    - The de-facto monopoly of high-performance compute nodes (servers, desktops, laptops).
    - Generally *too complex* for embedded systems.
  - **ARM**:
    - A (formerly) reduced instruction set (RISC).
    - Many versions with many different features, but all entirely controlled by one company.
    - The de-facto monopoly of embedded systems.

Enter RISC-V (pronounced “risk-five”)

- A completely open ISA that is freely available to academia and industry.
- A real ISA suitable for direct native hardware implementation, not just simulation or binary translation.
- An ISA that avoids “over-architecting” for a particular microarchitecture style (e.g., microcoded, in-order, decoupled, out-of-order) or implementation technology (e.g., full-custom, ASIC, FPGA), but which allows efficient implementation in any of these.

Why develop a new ISA?

- Preliminary question: **Why use an existing commercial ISA (e.g. ARM, x86)?**
  - Commercial ISAs have large and widely supported software ecosystems, development tools and ported applications, documentation and tutorials.
  - For example, this version of PowerPoint is running on x86 and x86 alone…
- **However, there are significant disadvantages:**
  - Commercial ISAs are proprietary
  - Commercial ISAs are only popular in certain market domains
  - Commercial ISAs come and go
  - Popular commercial ISAs are complex
  - Commercial ISAs alone are not enough to bring up applications
  - Popular commercial ISAs were not designed for extensibility
  - A modified commercial ISA is a new ISA

The First RISC-V Hackathon in Israel

- **Dec. 26-27, 2019 @WD Office, Kfar Saba**
- **Hosted by Western Digital and Mellanox**

- **Who should Join?**
  - SW and HW developers, passionate about technology, that would like to add new innovation to the RISC-V community.
  - It's an opportunity to hack in a new breakthrough architecture in an innovative environment. Get an access to Western Digital and Mellanox expert, win great prizes and have a lots of fun!
- **Come to contribute to the RISC-V ecosystem, join us at the RISC-V Hackathon.**
  - https://hackathon.forms-wizard.co.il

RISC-V Basics

ISA Overview

- **The RISC-V Base Integer ISA:**
  - Two options: RV32I (32-bit) and RV64I (64-bit)
  - *Must be present in any implementations*.
  - RV32E is a small microcontroller subset and RV128I is a future 128-bit ISA.
- **Standard Instruction Set Extensions:**
  - M: integer multiply, divide, remainder
  - A: atomic memory operations
  - F: single-precision floating point
  - D: double-precision floating point
  - **G: All of the above ("IMAFD")**
  - C: compressed instructions
    - 16-bit encoding for frequently used instructions

RISC-V Registers

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5-7	t0-2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function Arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller

- **Unlike HLL like C or Java, assembly cannot use variables**
- **Assembly Operands are registers**
  - limited number of special locations built directly into the hardware
  - operations can only be performed on these!
- **RISC-V has 32 Registers of 32-bits each**
  - 32-bits is a word in RV32, 64-bits in RV64
  - Registers are called `x0-x31`
  - With the ABI, we'll give them more comprehensible names
- **Floating Point adds 32 floating point registers:** `f0, f1, … f31`

Base Instruction Formats

RISC-V has several base instruction formats:

- **R-Format**
  - For 3 register operations

    `OPR rd,rs1,rs2`
- **I-Format**
  - For immediate instructions

    `OPI rd,rs1,Immed-12`
- **S-Format (and B-Format)**
  - For store operations (and branches)

    `OPS rs1,rs2,Immed-12`
- **U-Format (and J-Format)**
  - For long immediates (and Jumps)

    `OPU rd,Immed-20`

`rd` – destination register
`rs1` – source register 1
`rs2` – source register 2
`Immed-12` – 12-bit immediate
`Immed-20` – 20-bit immediate

```
ADD    x4,x6,x8   # x4=x6+x8
```

```
ADDI   x4,x6,123  # x4=x6+123
LW     x4,8(x6)   # x4=Mem[8+x6]
```

```
SW     x4,8(x6)   # Mem[8+x6]=x4
BLT    x4,x6,loop # if x4<x6 loop
```

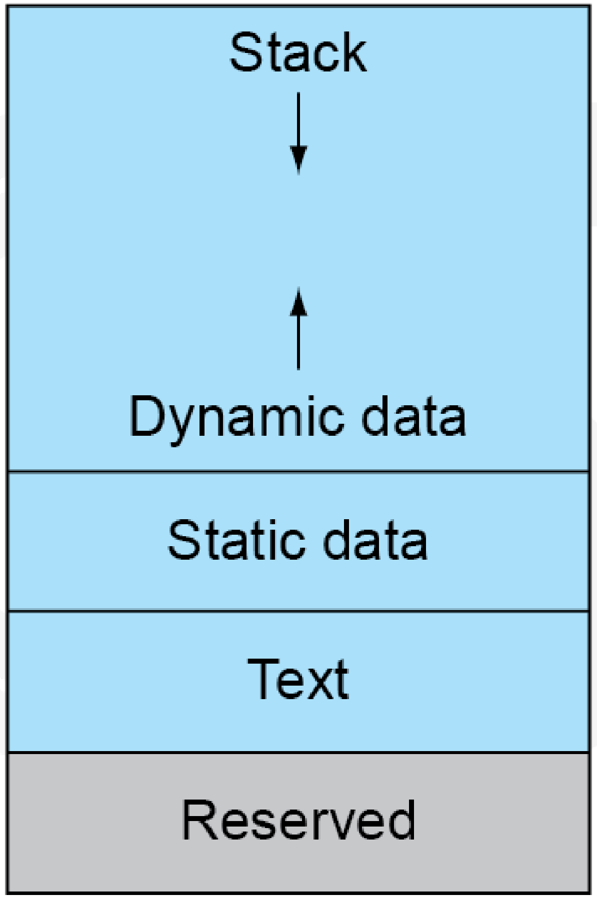
```
LUI    x4,0x12AB7 # x4=value<<12
JAL    x4,foo     # jump&link
```

Types of basic instructions

- **Arithmetic/Logical Operations**
  - ADD/SUB – register-register, register-immediate
  - AND/OR/XOR – register-register, register-immediate
  - SHIFT – right, left, logical, arithmetic, immediate, …
  - Comparison – set if less than register/immediate, signed/unsigned, …
- **Memory Operations**
  - Load from memory – load word, byte, halfword, doubleword, signed/unsigned
  - Store to memory – store word, byte, halfword, doubleword, signed/unsigned
  - Access 32-bit address: `LUI` (20 bit immediate) → `ADDI` (12 bit immediate)
- **Branch Instructions**
  - Conditional – Branch if equal, not equal, greater than, less than, …
  - Non conditional – Jump and link (PC relative), Jump and link register (absolute)

RISC-V Memory Map

- **Text:**
  - Program code
- **Static data:**
  - Global variables, e.g., static variables in C, constant arrays and strings
  - A global pointer (`GP`) is used initialized to address allowing ±offsets into this segment
- **Dynamic data:**
  - a.k.a., "**heap**"
  - e.g., `malloc` in C, `new` in Java
- **Stack:**
  - Automatic storage for managing procedure called



Source: P&H, Ch. 2

Calling Conventions

- **A procedure is initiated from within another piece of code.**
  - The initiating function is called the "caller" and the subroutine is the "callee"
- **In order to ensure that the caller's state is not changed during the subroutine, important data must be saved.**
  - The caller can save important registers before calling the subroutine.
  - The callee can save registers that are going to be overwritten during execution.
- **RISC-V calling conventions:**
  - The caller places the arguments in *argument registers* `a0`-`a7` (`x10`-`x17`).
  - The caller moves the *stack pointer* `sp` (`x2`) down.
  - The callee has to save the *saved registers* `s0-s11` (`x8`,`x9`,`x18`-`x27`)
  - The callee is allowed to write over the *temp registers* `t0-t6` (`x5-x7`, `x28`-`x31`).
  - The caller puts the *return address* (`PC+4`) in the `ra` register (`x1`)

So how is a procedure called and returned?

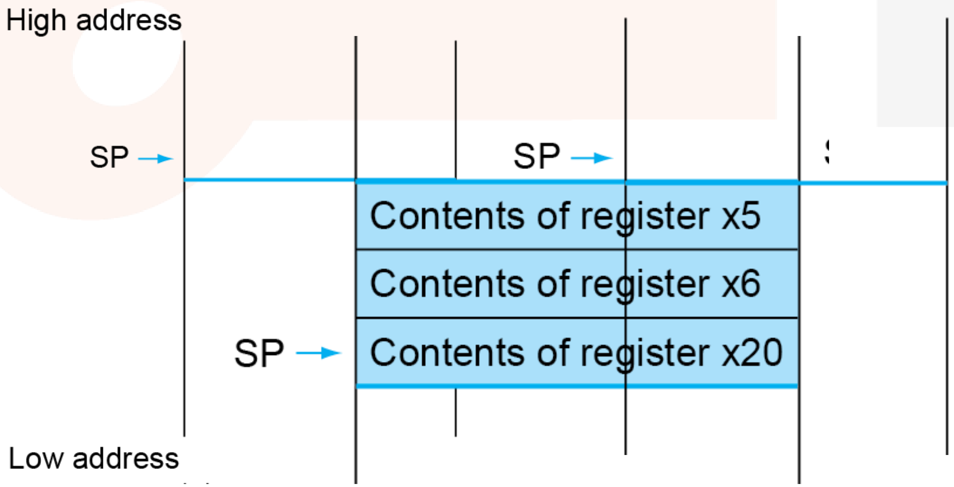
- **Calling a procedure (caller):**
  - Store the arguments in `a0-a7`.
  - If needed, move the stack pointer (`sp`) and store arguments on `stack`.
  - Update `PC` to the procedure address and save `PC+4` in `ra` (`JAL` command)
- **Running a procedure (callee):**
  - Use arguments stored in `a0-a7` or on the `stack`.
  - If callee needs to use `s0-s11`, move `sp` and store on `stack`.
- **Returning from a procedure (callee):**
  - Store return values in `a0-a1` or in `memory`.
  - Restore saved `s0-s11` (pop from `stack`) and update `sp`.
  - Give a `RET` command (`JALR x0, x1, 0`)
- **If caller saved any registers on stack, restore them upon returning.**

Example of Procedure Call

- **C code:**

```
long long int leaf_example (
   long long int g, long long int h,
   long long int i, long long int j) {
       long long int f;
       f = (g + h) - (i + j);
       return f;
}
```

- Arguments `g, …, j` in `a0, …, a3`
- `f` in `s1`
- temporaries `s2`, `s3`
- Need to save `s1, s2`, `s3` on stack



- **RISC-V code (64-bit):**

```
leaf_example:
    ADDI sp,sp,-24       Allocate Stack
    SD   s2,16(sp)       Save x5, x6, x20 on stack
    SD   s3,8(sp)
    SD   s1,0(sp)
    ADD  s2,a0,a1        s2 = g + h
    ADD  s3,a2,a3        s3 = i + j
    SUB  s1,s2,s3        f = s2 – s3
    ADDI a0,s1,0         copy f to return register
    LD   s1,0(sp)        Restore s1, s2, s3 from stack
    LD   s3,8(sp)
    LD   s2,16(sp)
    ADDI sp,sp,24        Deallocate Stack
    JALR zero,0(ra)      Return to caller
```

Source: P&H, Ch. 2

The RISC-V Software Toolchain

CALL: Compiling, Assembling, Linking and Loading

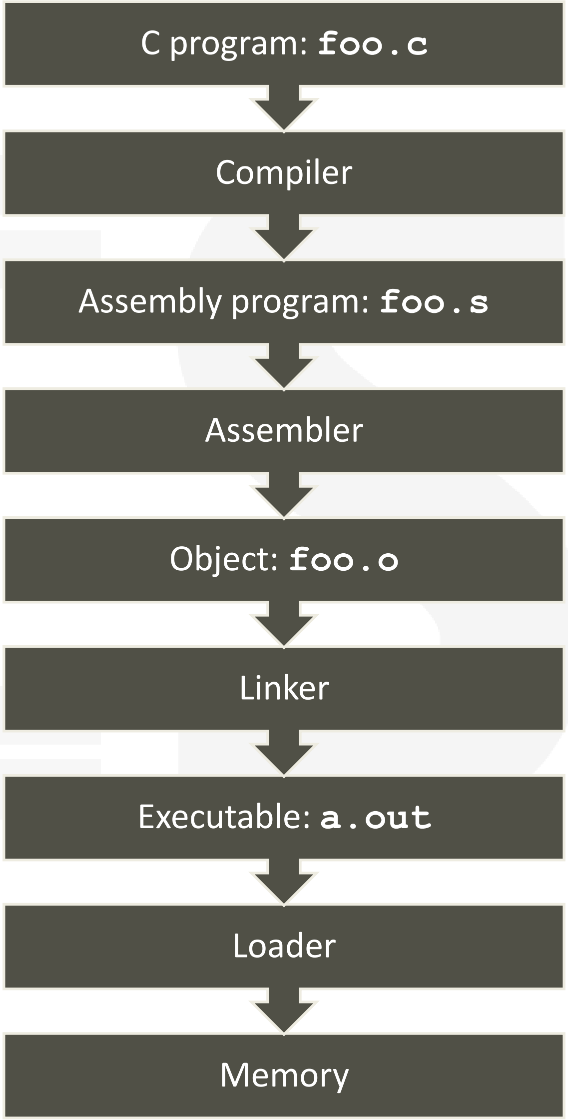
Steps in Compiling and Running a C Program

`gcc -O2 -S -c foo.c`

foo.c

- **Compiler**
  - Input: High-Level Language Code (`foo.c`)
  - Output: Assembly Language Code (`foo.s`)
  - (Note: Output *may* contain pseudo-instructions)

foo.s



Compiled `Hello.c`: `Hello.s`

```c
#include <stdio.h>
int main() {
   printf("Hello, %s\n",
                 "world");
   return 0;
}
```

```
.text                           # Directive: enter text section
  .align 2                      # Directive: align code to 2^2 bytes
  .global main                  # Directive: declare global symbol main
main:                           # label for start of main
  ADDI sp,sp,-16                # allocate stack frame
  SW   ra,12(sp)                # save return address
  LUI  a0,%hi(string1)          # compute address of
  ADDI a0,a0,%lo(string1)       #   string1
  LUI  a1,%hi(string2)          # compute address of
  ADDI a1,a1,%lo(string2)       #   string2
  CALL printf                   # call function printf
  LW   ra,12(sp)                # restore return address
  ADDI sp,sp,16                 # deallocate stack frame
  LI   a0,0                     # load return value 0
  RET                           # return
.section .rodata                # Directive: enter read-only data section
  .balign 4                     # Directive: align data section to 4 bytes
  string1:                      # label for first string
    .string "Hello, %s!\n"      # Directive: null-terminated string
  string2:                      # label for second string
    .string "world"             # Directive: null-terminated string
```

Steps in Compiling and Running a C Program

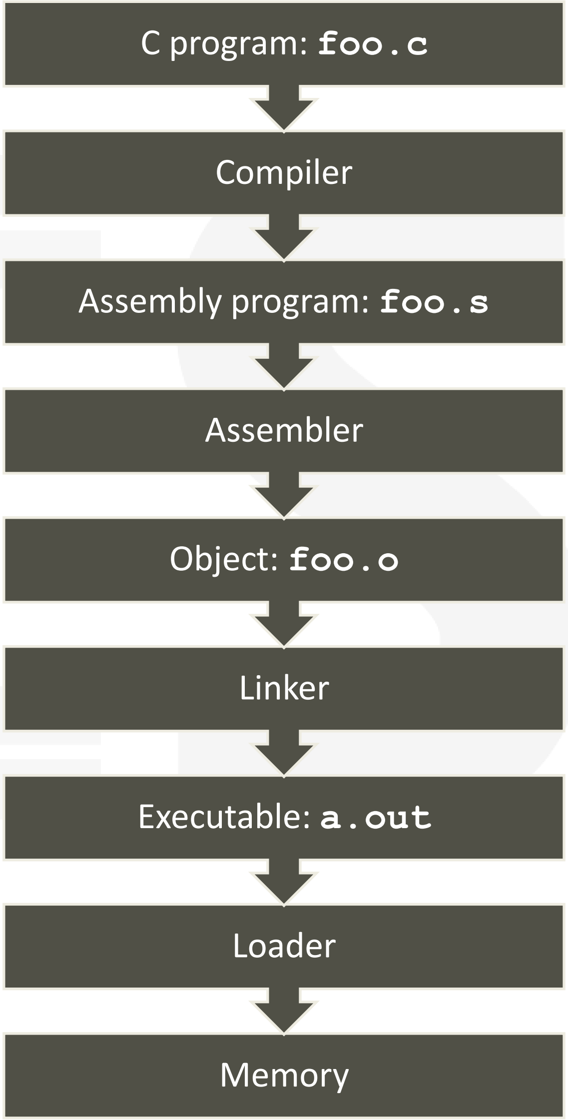
`gcc -O2 -S -c foo.c`

- **Compiler**
  - Input: High-Level Language Code (`foo.c`)
  - Output: Assembly Language Code (`foo.s`)
  - (Note: Output *may* contain pseudo-instructions)
- **Assembler**
  - Input: Assembly Language Code (`foo.s`)
  - Output: Object Code, information tables (`foo.o`)
  - Reads and Uses Directives
  - Replace Pseudo-instructions
  - Produce Machine Language
  - Creates Object File

`foo.c`

`foo.s`

`foo.o`



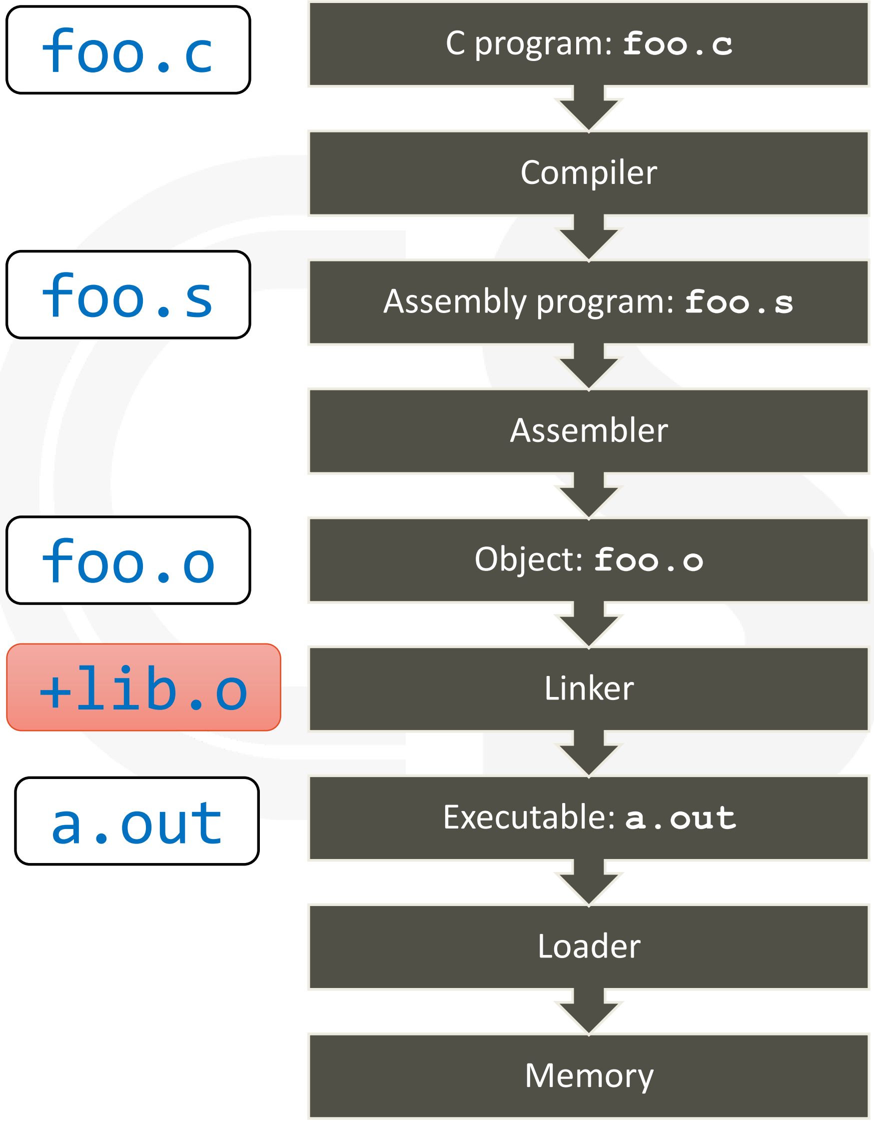
Assembling Hello.s → Linkable Hello.o

```
.text                       # Directive: enter text section
  .align 2                  # Directive: align code to 2^2 bytes
  .global main              # Directive: dec
main:                       # label for star
  ADDI sp,sp,-16            # allocate stack
  SW   ra,12(sp)            # save return ad
  LUI  a0,%hi(string1)      # compute addres
  ADDI a0,a0,%lo(string1)   #   string1
  LUI  a1,%hi(string2)      # compute addres
  ADDI a1,a1,%lo(string2)   #   string2
  CALL printf               # call function
  LW   ra,12(sp)            # restore return
  ADDI sp,sp,16             # deallocate sta
  LI   a0,0                 # load return va
  RET                       # return
.section .rodata            # Directive: ent
  .balign 4                 # Directive: ali
  string1:                  # label for first string
    .string "Hello, %s!\n"# Directive: null-terminated string
  string2:                  # label for second string
    .string "world"         # Directive: null-terminated string
```

```
00000000 <main>:
0:   ff010113 ADDI  sp,sp,-16
4:   00112623 SW    ra,12(sp)
8:   00000537 LUI   a0,0x0      # addr placeholder
c:   00050513 ADDI  a0,a0,0     # addr placeholder
10:  000005b7 LUI   a1,0x0      # addr placeholder
14:  00058593 ADDI  a1,a1,0     # addr placeholder
18:  00000097 AUIPC ra,0x0      # addr placeholder
1c:  000080e7 JALR  ra          # addr placeholder
20:  00c12083 LW    ra,12(sp)
24:  01010113 ADDI  sp,sp,16
28:  00000513 ADDI  a0,a0,0
2c:  00008067 JALR  ra
```

Steps in Compiling and Running a C Program

- **Linker**
  - Input: Object code files, information tables (e.g., `foo.o`, `libc.o`)
  - Output: Executable code (`a.out`)
  - Combines several `.o` files into a single executable
  - Enables separate compilation of files
    - Changes to one file do not require recompilation of the whole program (e.g., Linux source > 20 M lines of code!)
- **Loader**
  - Input: Executable Code (`a.out`)
  - Output: Code is loaded into memory
    - In practice, the loader is the Operating System (OS)



Linking `Hello.o` → Executable `Hello.out`

```
00000000 <main>:
0:   ff010113 ADDI  sp,sp,-16
4:   00112623 SW    ra,12(sp)
8:   00000537 LUI   a0,0x0     # addr placeholder
c:   00050513 ADDI  a0,a0,0
10:  000005b7 LUI   a1,0x0
14:  00058593 ADDI  a1,a1,0
18:  00000097 AUIPC ra,0x0
1c:  000080e7 JALR  ra
20:  00c12083 LW    ra,12(sp)
24:  01010113 ADDI  sp,sp,16
28:  00000513 ADDI  a0,a0,0
2c:  00008067 JALR  ra
```

```
000101b0 <main>:
  101b0: ff010113   ADDI sp,sp,-16
  101b4: 00112623   SW   ra,12(sp)
  101b8: 00021537   LUI  a0,0x21
  101bc: a1050513   ADDI a0,a0,-1520 # 20a10 <string1>
  101c0: 000215b7   LUI  a1,0x21
  101c4: a1c58593   ADDI a1,a1,-1508 # 20a1c <string2>
  101c8: 288000ef   JAL  ra,10450    # <printf>
  101cc: 00c12083   LW   ra,12(sp)
  101d0: 01010113   ADDI sp,sp,16
  101d4: 00000513   ADDI a0,0,0
  101d8: 00008067   JALR ra
```

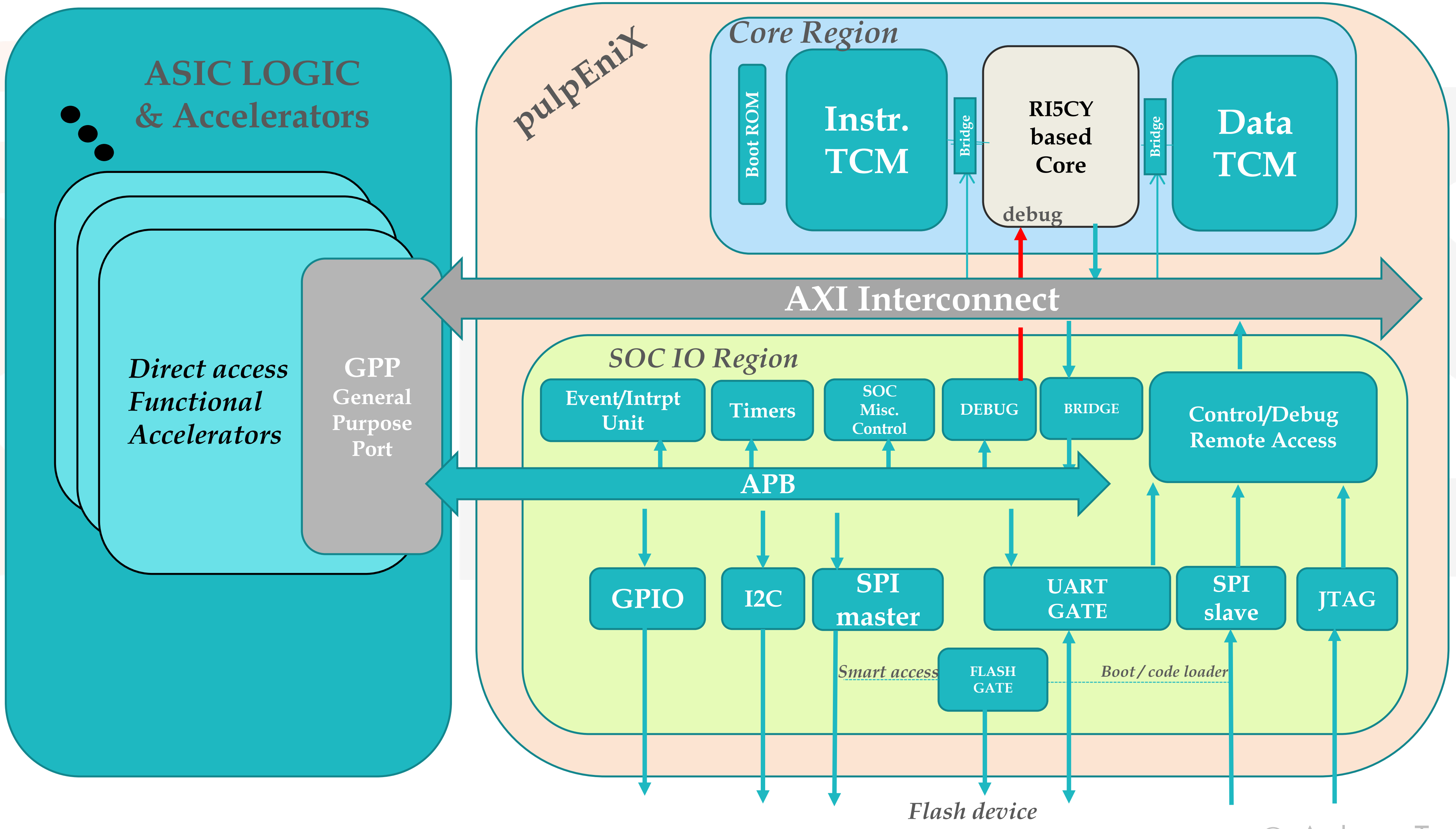
Introduction to PulpEnIX

PulpEniX

- PulpEnIX (nickname for PULP-Enics) is EnICS SoC/RISC-V research platform. Forked from open-source PULP platform https://www.pulp-platform.org/

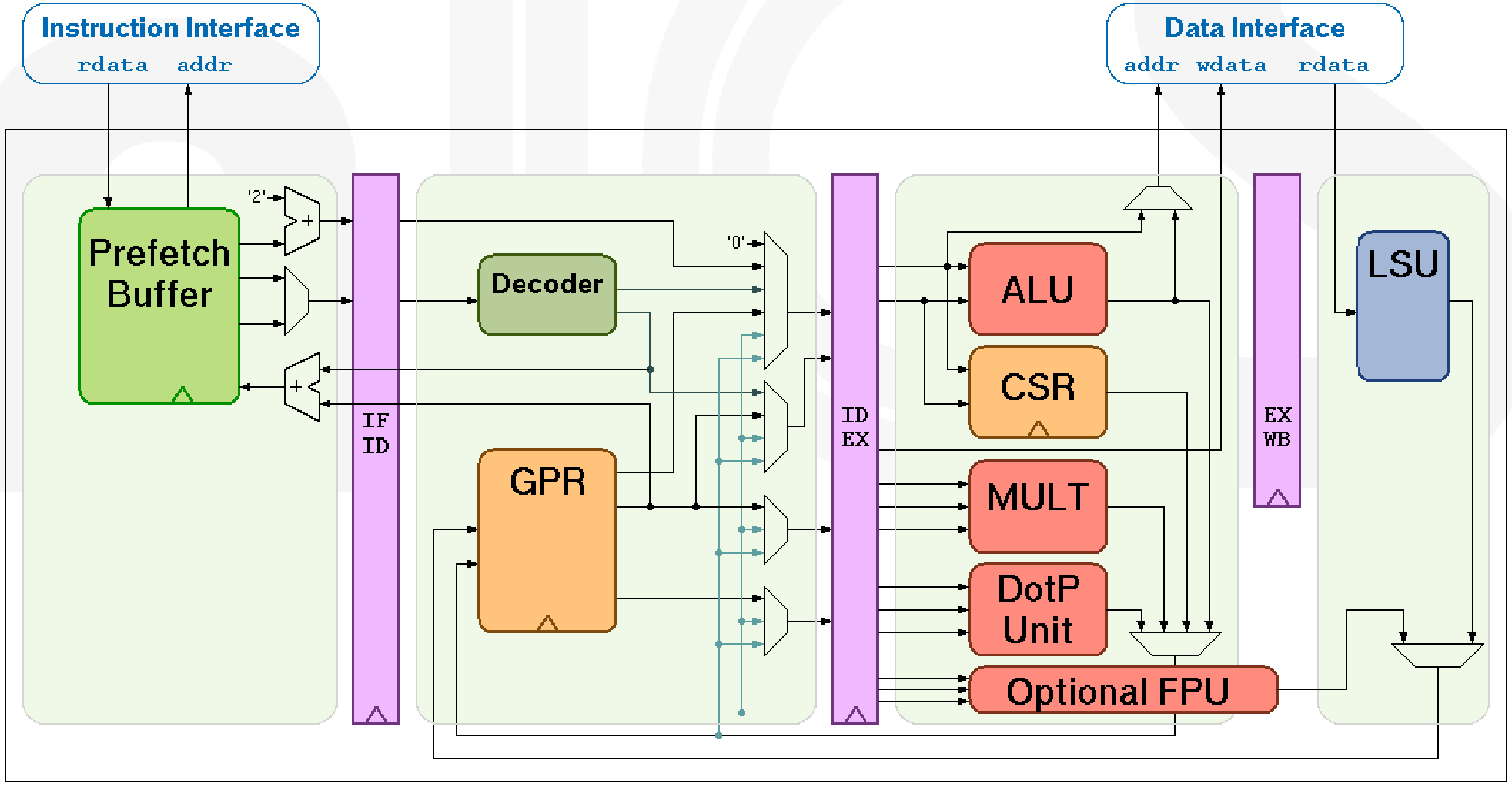
- Embedded SOC oriented
- HW/SW (Hardware/Software) co-design environment.
- Simulation environment.
- FPGA platform environment.
- Embedded RISC-V enhancements workbench.

Pulpenix SOC Perspective



The RI5CY Core

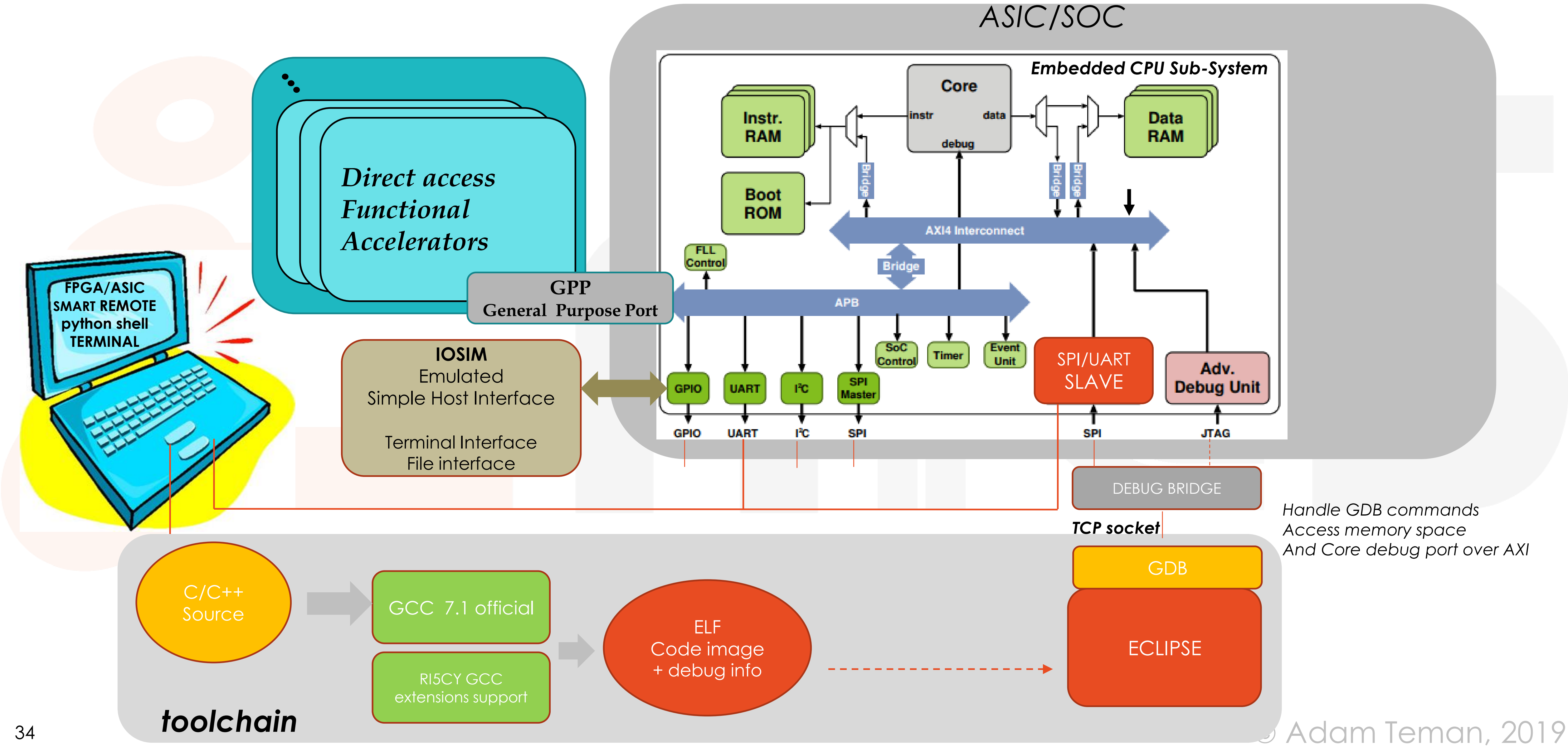
- **RI5CY is a 4-stage, in-order 32b RISC-V processor core.**
- **The ISA of RI5CY was extended to support additional instructions including:**
  - Hardware loops
  - Post-increment load and store instructions
  - And additional ALU instructions that are not part of the standard RISC-V ISA.



PulpEniX HW/SW development Environment

- **Toolchain:**
  - A complete GCC based toolchain, including PulpV3 extensions support
- **PulpEniX C libs**
  - A set of simple libraries for files and terminal IO functionality
- **PulpEniX simulation environment**
  - Verilog based simulation environment. (cadence)
  - Optional Verilator and Modelsim references
- **PulpEniX FPGA platform environment**
  - FPGA Hardware and Software reference environment.
  - Smart host interface.

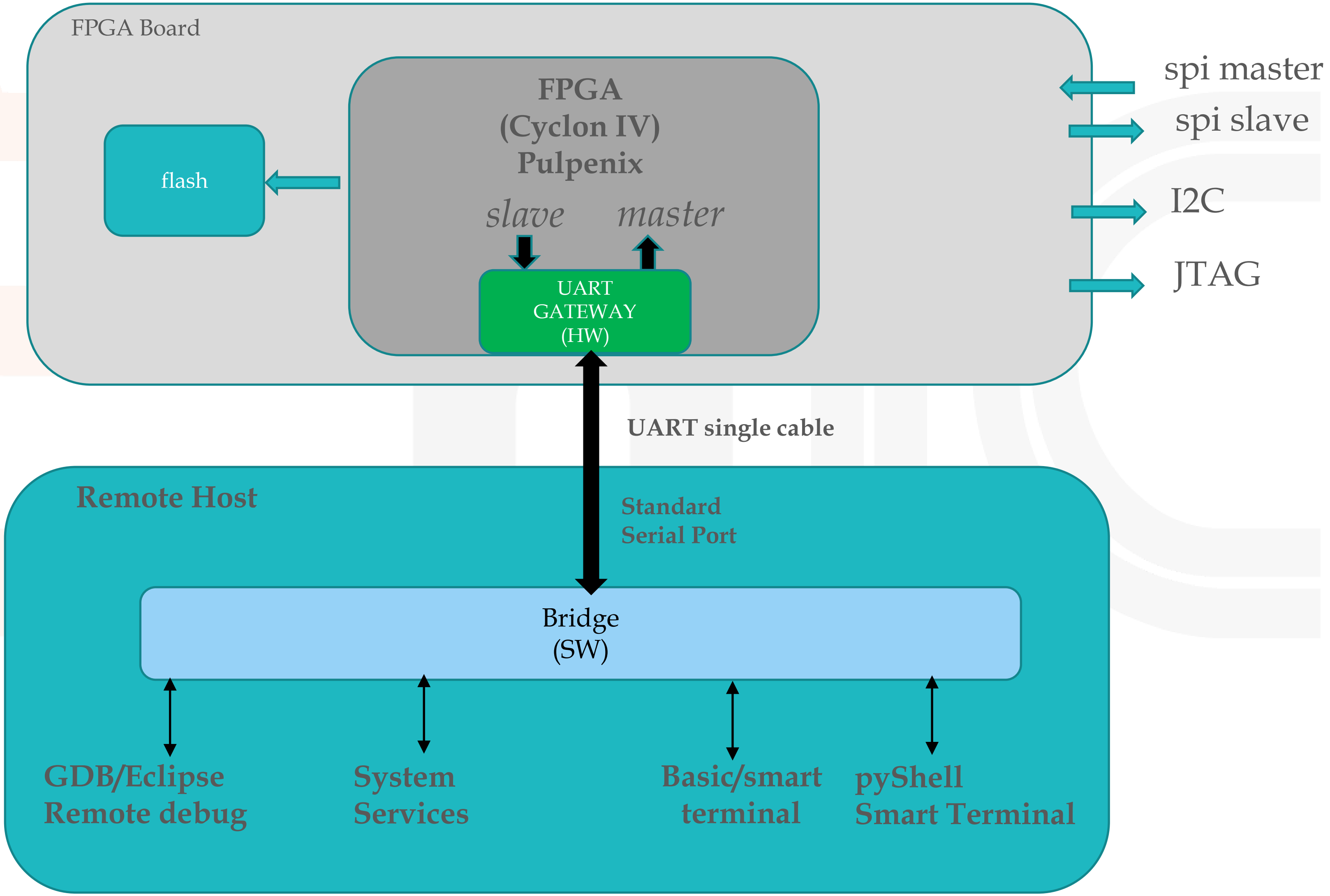
HW-SW Co-Design environment



PulpEniX FPGA Platform

- Based on Altera Cyclone IV , EP4CE115 cost effective ($85) board
- Simple direct unleashed IO interface
- Single cable smart-UART interface
- Smart python interface shell
- GDB and Eclipse based Debug
- File interface & Terminal stdio
- Code loader
- Remote AXI/APB address space access
- Remote access over SPI
- Convenient design verification platform

FPGA board smart-UART remote access



RISC-V Hackathon orientation: PulpEniX platform target

- **Open for a wide range of proposals which utilize or enhance the platform Including but not limited to one or more of the following:**
- **Software application utilizing the platform**
  - Example: code and run on the platform Conway's Game of Life , algorithm will run on the platform, animation will run on the host computer connected to the platform.
- **Hardware enhancement to the platform/core**
  - Example: develop, integrate and demonstrate utilization of new hardware implemented ALU instructions such as a finding the average of two operands
- **Developing and integrating a Hardware accelerator**
  - Example: develop a hardware Huffman-Code compression/decompression utility interface with the accelerator utilizing the APB GPP (General Purpose Port)
- **Interfacing the platform with external devices**
  - Example: Interface PulpEniX (over GPIO pins) with a temperature sensor and a fan , control the fan to maintain ambient temperature near a heat source.

RISC-V Hackathon-
Schedule and support PulpEniX platform target

- **Projects registration by 15/Nov**
- **Announcement of the projects selected 1/Dec**
- **PulpEniX platform training for participants at BIU Week of 8/Dec** (not final)
    - Explain in detail and exercise the platform with a reference project.
- **Early platform delivery to participants**
    - Possible up to 2 weeks prior to the event, subject to project complexity.
    - No limitation on early use of the simulation environment.
- **The RISC-V Hackathon event at WD Kfar-Saba 26-27/Dec**

PulpEniX Demo

- Toolchain & Compiling a program
- Loading a code image
- Optional Menu Program interface
- Dumb Terminal usage
- pyshell usage
- Executing a program
- Debugging a program using Terminal GDB
- Using Eclipse to Debug a program
- Coremark

How to find us?

- **The EnICS Team is available to help you develop your idea for the RISC-V Hackathon.**
- **You can contact us at:**
  - Dr. Adam Teman: adam.teman@biu.ac.il
  - Udi (Yehuda) Kra: yehuda.kra@biu.ac.il
  - Yonatan Shoshan: yonatan.shoshan@biu.ac.il
  - Yehuda Rudin: yehuda.rudin@biu.ac.il
  - Tzachi Noy: tzachi.noy@biu.ac.il

- **Good Luck!**

EnICS
Emerging Nanoscaled
Integrated Circuits and Systems Labs